Handbook for the Computer Security Certification of Trusted Systems

NRL Technical Memorandum 5540:080A, 24 Jan 1995

Naval Research Laboratory, Code 5540, Washington, D.C. 20375-5337

For additional copies of this report, please send e-mail to
landwehr@itd.nrl.navy.mil, or retrieve PostScript via
http://www.itd.nrl.navy.mil/ITD/5540/publications/handbook
(e.g., using Mosaic).

The Security Policy Model:
A Chapter of the Handbook for the Computer Security Certification of Trusted Systems

C.N. Payne
Center for High Assurance Computing Systems
Code 5542
Naval Research Laboratory
Washington, D.C.

Contents

1 Introduction

The *security policy model* (SPM) is the apex of the assurance argument in the development of a trustworthy system. A system is trustworthy if the probability of a catastrophic flaw remaining after testing and review is acceptably low [34]. In our case, the correct behavior is described informally in a security policy. The SPM presents the constraints of the security policy in a way that can be analyzed formally. It guides the developer toward a secure design and implementation and helps the developer demonstrate rigorously that the system enforces its security policy. The SPM describes the system's security-relevant behavior for the user. Without the SPM, the security-relevant goals of the system are not well-defined for the developer, the user, or the evaluator. The SPM expresses the definition of security for a trusted system.

In this chapter, we will instruct the evaluator in the assessment that an SPM fulfills the purposes identified above. We will identify the structure of an SPM and guide the evaluator in the analysis of this structure. The SPM is influenced heavily by the critical requirements of the trusted system. These requirements determine the system entities that must be modeled and the relationships between those entities. We will discuss one way to determine if the appropriate entities and relationships have been modeled and if the critical requirements on those entities have been satisfied.

We will *not* try to determine what constitutes a good definition of security for a trusted system. Much research has been performed already regarding good definitions of security. Significant coverage of this research may be found in the work of Landwehr [17], Millen [27], Williams [32], and McLean [25]. Rather than repeat what has already been summarized effectively, we will concentrate on the evaluation of the structures in the SPM that support the definition of security.

This chapter is divided into six sections. In the second section, "An Evaluator's Overview", we present the attributes of a good SPM and its purposes for the developer, the evaluator, and the user. We also identify the resources for evaluating an SPM.

In the third section, "Understanding Modeling", we discuss the task of modeling beyond the limited context of security. Here the parts of the model are identified, and the qualities of effective communication are discussed.

In the fourth section, "Modeling Security", we consider how the model's elements are interpreted for security and what role they play for assurance. The discussion in this section forms the groundwork for the SPM's evaluation.

In the fifth section, "The Trusted Product SPM", we examine the modeling issues for a trusted system that is based on a trusted product.

Finally, the sixth section, "The Evaluation", presents a set of guidelines for evaluating the SPM. Not intended as a checklist, this section offers motivating questions for the evaluator.

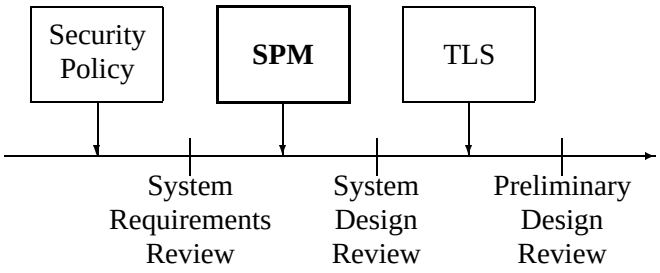

Figure 1: SPM in the Development Life Cycle

2 Overview of the SPM

This section summarizes for the evaluator the characteristics of an acceptable SPM, including answers to some commonly asked questions, a discussion of how the SPM is produced and an elaboration of the purposes it serves for the developer, the user and the evaluator. The modeling exercise itself will be discussed more thoroughly later.

2.1 Commonly Asked Questions

What is an SPM?

The SPM is a mathematical restatement of the security policy that must be enforced by the computer system. It provides the *definition of security* for the computer system. It is used to analyze the policy for consistency and to clarify the definition of security of the trusted system for the user, the developer and the evaluator.

Why do I need it?

The SPM plays a crucial role in the construction of the assurance argument that the system enforces the policy, because it identifies constraints on the behavior of the system and can be subjected to formal analysis.

When is it developed?

The SPM is delivered early in the development life cycle. It is written after the security policy is complete, and before the top-level specification (TLS) of the design is approved. A typical timeline for security policy, SPM and TLS completion is illustrated in Figure 1. Each vertical hash in the timeline represents a milestone in the system development, e.g., the security policy is completed before the system requirements review. The names of these milestones may vary.

One of the challenges facing every trusted system development is the proper integration of system engineering and security engineering. Too often a developer adopts two parallel development paths: one for the security engineering deliverables and one for the system engineering deliverables. This approach is rarely successful. The two development teams diverge

quickly because they must satisfy different standards: a security standard (TCSEC) and a software development standard (*DOD-STD-2167A Defense System Software Development* [6]). The software development standard enumerates the documents and other products to be delivered during the development of an automated data processing system, but it does not acknowledge the security deliverables required by the TCSEC. Consequently, if the system engineering team proceeds without knowledge of the assurance requirements mandated by the TCSEC, the trusted system may fail evaluation. Instead, the developer should proceed carefully with an integrated engineering team.

Finally, the SPM should be maintained along with the system. While Figure 1 illustrates a waterfall approach to system development, in reality the SPM may be updated even after its acceptance at System Design Review. The SPM is much more than just a deliverable in a product lifecycle. It should be the keystone of what is enforced by the system. It should be revisited at each stage of the development lifecycle, and it should be reinterpreted, at least informally, for each refinement of the design.

What resources must be devoted to its evaluation?

The evaluation of the SPM is not a trivial task. The time and effort to be allocated to the SPM and all other tasks are discussed in the handbook chapter on certification. Another good source for guidance on trusted system procurement is [7]. The discussion below focuses on the human resource — the specific qualities of the SPM evaluator.

An evaluator should understand the evaluation criteria, i.e., the TCSEC, and its modeling requirements thoroughly. The evaluator must be prepared to interpret the criteria's requirements for the trusted system. Much guidance exists for understanding the TCSEC's modeling requirements, e.g., [32] and [27], but this chapter is among the first to describe how these requirements apply to trusted systems.

The evaluator should understand the security policy and the overall mission thoroughly. The evaluator must recommend clarifications in the security policy whenever the SPM suggests ambiguities. The evaluator will often be an arbiter between the developer and the customer over security policy interpretation. The evaluator should also be aware of the system architecture assumptions that underlie the development of the SPM, in order to assess whether the SPM is complete.

The evaluator should be well grounded in mathematics and formal methods. It would be helpful for the evaluator to be familiar with automated specification and verification tools. Although these tools are not required for all developments, e.g., TCSEC developments targeted below A1, the rigor they impose is valuable in most developments. The evaluator should feel comfortable reading the SPM's mathematical notation without an English translation.

2.2 The Security Policy

The statement that the SPM is a 'model of the security policy' is altogether too vague since the term 'security policy' can mean many things. We view the security policy that is modeled as the *automated security policy* [39]. Figure 2 illustrates how the security policy for the computer (the COMPUSEC policy) is derived.

National security policy objectives influence the security policy that is adopted by an organization, and the latter policy, along with the goals of the mission and the known threats to that

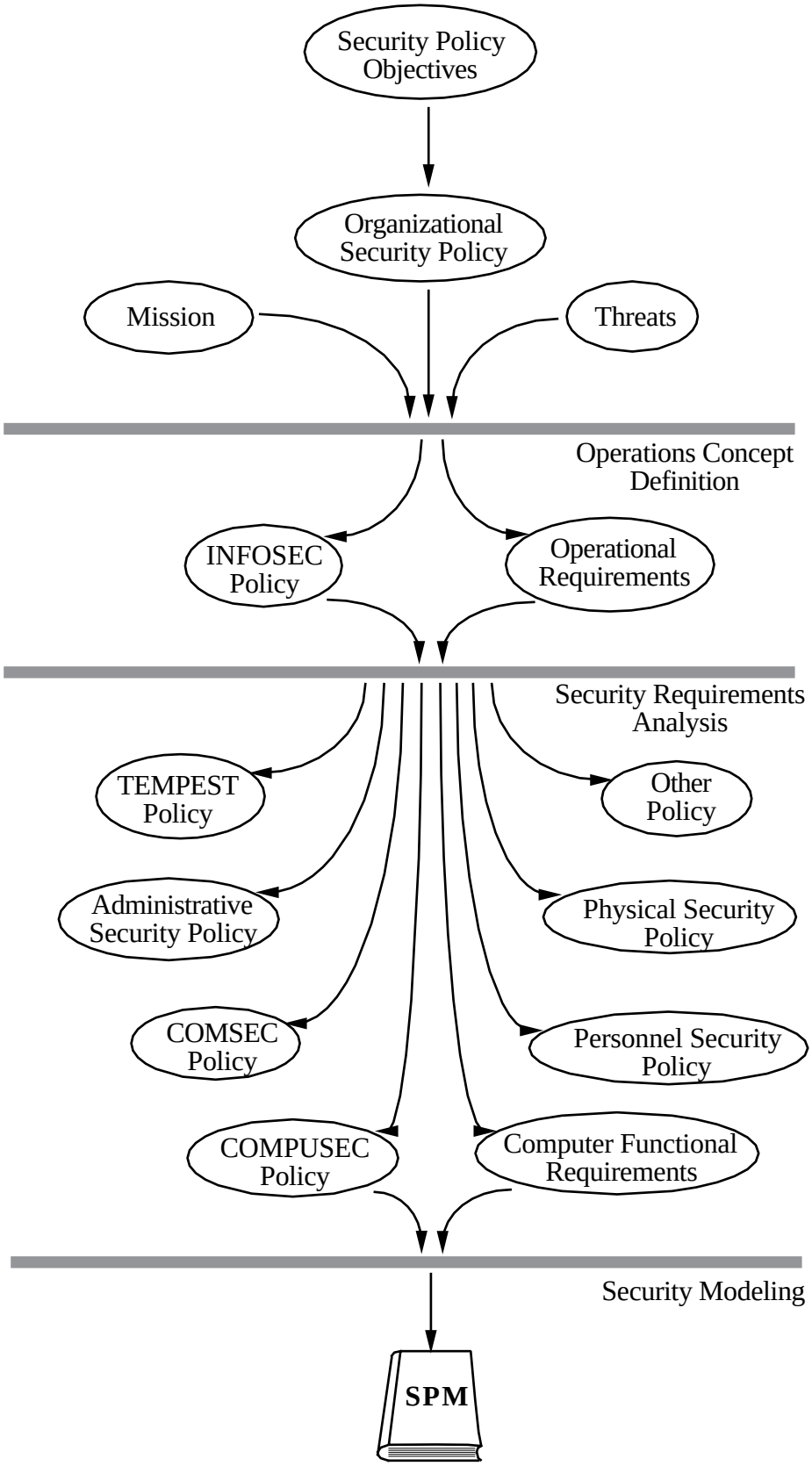

Figure 2: The ‘security policy’ that is modeled

mission, determine the *information security (INFOSEC) policy* (as discussed by Payne [35]) and operational requirements for the mission. The INFOSEC discipline and its policy can be divided into several subdisciplines — each with its own policy concerns. They include

- **COMPUSEC** – computer security requirements, i.e., constraints on the computer hardware and software,
- **COMSEC** – communications security requirements, i.e., constraints on the communications between the computer system and its environment,
- **TEMPEST** – hardware device emanation controls,
- **Physical Security** – physical structure controls on the buildings that house the computer system, e.g., locks on the doors,
- **Adminstrative Security** – controls of administrative practices, e.g., so that the correct operation of the computer system is ensured,
- **Personnel Security** – controls on individuals, e.g., clearance requirements, and
- **Other** – miscellaneous controls not described above.

Each discipline contains a set of countermeasures that will be used to enforce the INFOSEC policy. For example, the trusted system is the primary countermeasure for the COMPUSEC discipline. For each discipline, a set of assertions describes the behavior of the countermeasures and a set of assumptions describes the conditions upon which the assertions are based. A relationship exists between the disciplines such that the assumptions for one discipline are characterized as assertions for another discipline. If they do not, a vulnerability exists. If a threat exposes that vulnerability, then the system is at risk. Figure 3 illustrates the assumptions and assertions tradeoff for a simple example. The two unsupported assumptions (one for the COMPUSEC discipline and one for the administrative security discipline) represent security vulnerabilities.

The COMPUSEC discipline embodies most of the requirements to be enforced solely by the computer system. The COMSEC discipline may include some computer system requirements as well, but often those are evaluated separately from the COMPUSEC requirements. The SPM expresses the COMPUSEC assertions and assumptions. Together the assertions and assumptions constitute the definition of security for the trusted system.

The developer should provide both informal and formal descriptions of the SPM. The formal description is required for systems targeted for TCSEC class B2 or above. The informal description should include the assumptions, plus the user's view of the trusted system and an informal statement of the assertions. The formal description should define a computational framework expressed in some mathematical notation, and the assertions should be restated in that framework. The SPM should be supported by an assurance argument that the definition of security is correct and complete.

2.3 Creating the SPM

The process for creating the SPM is illustrated in Figure 4. The developer considers the effect of the COMPUSEC policy on the system's functional requirements and derives a set of

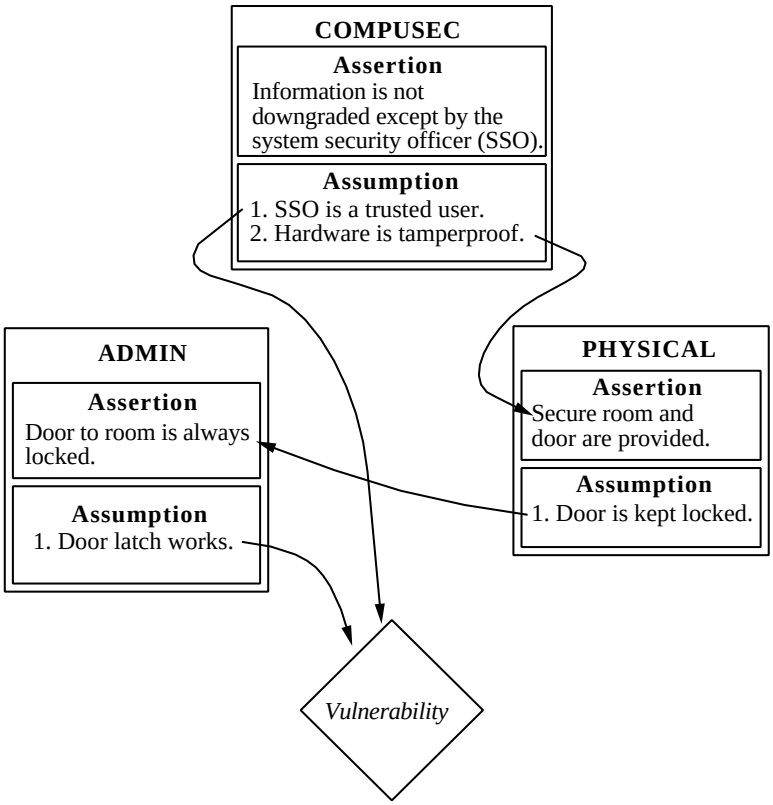

Figure 3: Security Discipline Interaction

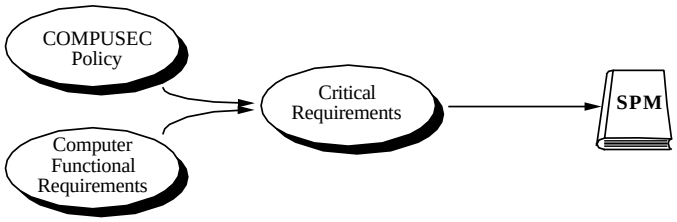

Figure 4: SPM Development

security-critical requirements. The critical requirements state in effect, "Given that a system with these functional requirements must enforce this security policy, how must the system's functional behavior be constrained so that the policy will be enforced?" The critical requirements are the assertions and assumptions for the trusted system. The SPM developed under this approach provides a definition of security that is valuable as a design tool and an assurance tool, because it is expressed in terms of the system's operational requirements. The Secure Military Message System (SMMS) SPM [16] used this approach. Its authors felt that keeping the application in mind during the modeling effort was very important [18]. Other inputs to the SPM that are not illustrated in Figure 4 include architectural assumptions about the implementation, a source for the SPM's mathematical notation and the TCSEC's modeling requirements.

The computational framework for the SPM is based on assumptions about the architecture of the completed system. The developer wants to minimize the effort required to interpret the SPM for the top-level system specification. The choice of computational framework may vary

depending on whether the intended architecture is, for example, distributed, parallel or monolithic. If these architectural assumptions are not considered, the interpretation of the SPM for that architecture may prove impractical.

The developer will probably use an existing formal notation for the formal description of the SPM. This notation should be grounded formally, but the foundations do not need to appear in the SPM. Instead, the SPM's bibliography should contain a reference to the notation's supporting documentation.

The TCSEC imposes only a few requirements on the SPM. For classes B2, B3 and A1, the TCSEC requires a formal description and states that the SPM shall be "proven that it is sufficient to enforce the security policy". [31, p. 32] The developer must demonstrate that the SPM restates the COMPUSEC policy completely and accurately. Because the COMPUSEC policy is an informal document, this proof will be informal.

The TCSEC also states that the SPM "shall be maintained over the life cycle of the ADP [Automated Data Processing] system" and it shall be "proven consistent with its axioms." [31, p. 31][1] The latter requirement is vague unless we understand what the TCSEC authors intend by the term "axioms". A probable explanation can be found in the TCSEC's glossary, under the entry for *Formal Security Policy Model*[2]:

> A mathematically precise statement of a security policy. To be adequately precise, such a model must represent the initial state of a system, the way in which the system progresses from one state to another, and a definition of a "secure" state of the system. To be acceptable as a basis for a TCB, the model must be supported by a formal proof that if the initial state of the system satisfies the definition of a "secure" state and if all assumptions required by the model hold, then all future states of the system will be secure. [31, p. 113]

So, "axioms" may be interpreted as the security properties that define the secure state.

2.4 Purposes of the SPM

This section reviews the purposes of the SPM for the developer, the user and the evaluator.

2.4.1 Developer

The developer reaps the greatest benefit from the definition of the SPM. There are four ways that the SPM serves the developer.

- **Clarify the definition of security.** The SPM helps the developer answer an important question: "Is the user's security policy consistent?". By stating the definition of security in a formal, computational framework, the developer can determine if the security policy is mathematically consistent.

[1] There is confusion over the meaning of the phrases "proven consistent with its axioms" and "proven that it is sufficient to enforce the security policy". Most experts agree that the transition rules must be demonstrated via formal proof to preserve the definition of security but that a rigorous mapping will suffice for a "proof" that the SPM enforces the security policy. More about this later.

[2] In other words, the requirements implied by this definition refer mainly to the formal description of the SPM.

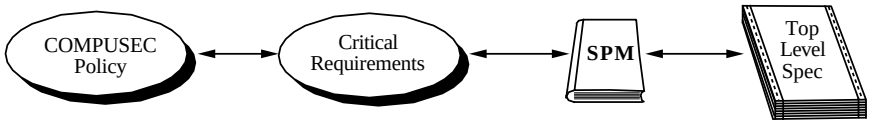

Figure 5: Partial View of the Assurance Argument

- **Guide the Design and Implementation.** While the SPM does not describe the design of the system, it does define constraints on the behavior that any design may exhibit. These constraints, when interpreted for the concrete system structures specified in the top-level design specification, provide guidance for the correct, secure design of those structures.

- **Provide Context for Security Testing.** The SPM provides a context for security testing by identifying the abstract entities and operations of the system that are security-critical. Through the SPM interpretation, the developer can identify the data structures and procedures that correspond to these abstract entities and target them for security testing. Also, the formal description of the definition of security plays a critical role in the development of the security test plan.

- **Contribute to the Assurance Argument.** The assurance argument is the convincing demonstration that the system enforces its security policy. It is constructed by associating each level of specification with the next lower and the next higher levels. At the top are the critical requirements, and at the bottom is typically the source code for the system. (The assurance argument is discussed more thoroughly in the Handbook chapter on Assurance Mappings.) A partial view of the assurance argument is illustrated in Figure 5. As development proceeds from the COMPUSEC policy to the top-level specification, the developer must demonstrate that each new level can be derived from the previous one. The vectors in Figure 5 illustrate that each argument is two-way, e.g., the SPM must restate all of the critical requirements and only those requirements.

2.4.2 User

The user's definition of security for the trusted system is expressed informally in the COMPUSEC policy. However, it is difficult to analyze the policy rigorously. Formally restating the policy in a computational framework facilitates this analysis and answers two questions for the user.

- "Does the developer understand my definition of security?" The user should compare the developer's definition with his own. If the developer has misinterpreted the user's intentions, any misunderstandings can be eliminated before costly design mistakes are implemented.

- "Is this really what I want?" In other words, is the definition of security reasonable? The security policy may appear reasonable, but when it is restated precisely in a computational framework, it may not be what the user intended.

It is critical to the development that both answers are affirmative.

2.4.3 Evaluator

For the developer, the SPM is a valuable design tool. For the user, the SPM clarifies and validates his definition of security. For the evaluator, the SPM is a early and reliable measure of the development's success. In particular, the evaluator is concerned with these issues:

- Do the user and the developer agree on the definition of security?
- Has the developer made a concerted effort to express the definition of security rigorously? Has he omitted important details? Does he understand the security policy?
- Does the developer appreciate the mechanisms necessary to enforce the definition of security? What are the potential pitfalls?

The SPM helps the evaluator determine if the developer appreciates his task. A half-hearted attempt at defining the SPM may indicate that the developer does not appreciate the rigor required for a trusted system development effort.

2.5 Important qualities for an SPM

An SPM's value is determined by how closely it represents the user's view, as expressed informally in the COMPUSEC policy, of the system's correct security behavior. A good SPM satisfies four important criteria: *completeness*, *correctness*, *consistency*, and *clarity*.

The SPM is *complete* if its definition of security restates all of the relevant assertions from the COMPUSEC policy. Some assertions may be omitted from the SPM. For example, the SPM might include all of the assertions relating to the prevention of security violations, but an audit requirement may be omitted because it addresses the detection of those violations. The evaluator and the developer can negotiate completeness. Typically, a mapping between the SPM and the COMPUSEC policy is enough to demonstrate completeness. This is a two-way mapping, i.e., all security assertions must be included in the SPM (with the possible exceptions noted above), and all assertions included in the SPM must be derived from the COMPUSEC policy.

The SPM is *correct* if its definition of security is an accurate statement of the relevant security assertions derived from the COMPUSEC policy. Correctness is the most difficult — and the most important — criterion to satisfy. The distinction between completeness and correctness is that in the former, all appropriate assertions are included, and in the latter, they are restated properly. While completeness can be negotiated, correctness cannot.

The SPM is *consistent* if it is not mathematically contradictory. An automated tool can check for consistency. The developer should also use the formal notation consistently, e.g., the null set should be represented consistently throughout the model.

The SPM is *clear* if it is specified simply and without extraneous detail. However, the SPM must include enough detail so that it is unambiguous. A clear specification simplifies the construction of the assurance argument. Clarity is particularly important for complex security assertions. It is crucial that all parties — the developer, the user and the evaluator — agree on the definition of security.

Satisfying these four criteria is not trivial. The developer should consider using formal methods. The specification language of a formal verification system is a useful tool because

it is well-defined and assists in the creation of clear and consistent structures. The verification system's analysis tools are useful for removing syntactic errors and minor semantic errors. The developer should use all methods available to reduce complexity in the SPM, because if the SPM is overly complex, the development of the trusted system is already at risk.

3 Understanding Modeling

The evaluation of an SPM for a trusted system must be based on objective evidence. One of the most difficult tasks for the evaluator is finding this evidence. In this section, we will explore modeling generally (i.e., beyond the context of security) in order to identify some basic criteria for evaluating a model. In the next section, we will interpret these criteria for the context of security modeling. The interpreted criteria will be the foundation for our evaluation of the SPM.

Intuitively, a model is an abstraction of a physical system that embodies some relevant behavior of that system for the purpose of analysis. For example, a city planner may want to model commuter traffic flow to simulate the effects of a proposed rail system. The model is useful if it has the characteristics of the actual traffic flow that are critical for analysis, e.g., the number of vehicles per hour on major arteries, but ignores details that are not so critical, such as the number of Toyotas versus the number of Fords. The model reduces the cost of analysis while maintaining predictable results.

The modeler must be very careful to represent all relevent entities in the model. For example, if the city planner does not consider the proposed roundtrip ticket cost to commuters for the rail system, then the modeler may overestimate the predicted rider base. To determine which entities of the problem domain are relevent, the modeler must understand the primary goals of the modeling exercise. In this example, if the primary goal is to determine how many commuters would rather take the train than drive, then the ticket price and the proximity of rail stations to major population centers become critical parameters.

The assumptions that underlie the model must also be identified. For example, the city planner may assume (erroneously) that commuters are generally unwilling to carpool. As a result, the planner may propose a smaller subsidy to the ticket price (i.e., the price is higher). Oblivious to this assumption, someone may misinterpret the model's results and believe that the ticket price reflects accurately the commuter's willingness to carpool. When reality demonstrates that commuters are *much* more willing to carpool, the single-ride cost will have be be lowered and the city will lose money. (The willingness to carpool should be a parameter of the model instead of an assumption.) An evaluator of the model should ensure that the assumptions are reasonable for the application.

In summary, a model is used to analyze the behavior of a physical system. It is a design tool that can help the developer avoid costly mistakes. If the model describes a behavior that is unacceptable, then the design can be changed inexpensively.

While this intuitive description of the model is helpful, it is not rigorous enough to identify and reason about the qualities and attributes of an acceptable model. For that, we turn to the discussion of modeling provided by Zeigler [40].

3.1 Basic elements

According to Zeigler, there are three major entities in the modeling universe:

1. the **real system**, a source of behavioral data,

2. the **model**, a set of instructions for generating the behavioral data of a real system, and

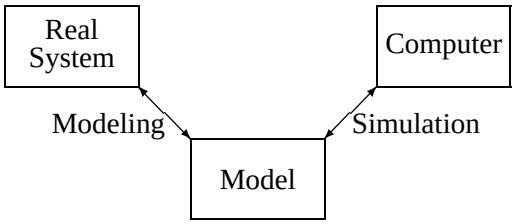

Figure 6: Modelling Universe

3. the **computer**, a computational process that can generate behavioral data when supplied with suitably encoded model instructions.

These entities and the relationships between them are illustrated in Figure 6.

The **real system** refers to "nothing more or less than a source of observable data." [40, p.28] The system may already exist, in which case its behavior can be observed easily, or it may not exist yet, in which case we predict its behavior from its requirements.

From the real system, we construct the **model** in three steps, as illustrated in Figure 7. First, we consider a *base model* of the real system. The base model describes completely the behavior of the real system. Unfortunately, it is too complex to write down (hence the dashed lines), so we must identify a simplified frame of reference, called the *experimental frame*, from which to observe the real system's behavior. The base model is the union of all possible experimental frames. The model that we actually write down, the *lumped model*, is valid only for the experimental frame that we specify. For our simple example, the base model is the entire transportation system, the experimental frame is our purpose for modeling the system (e.g., to determine the ridership of the proposed train), and the lumped model consists of those instructions necessary to generate the behavior of the system for the purpose identified. To summarize, the lumped model is a simplified base model for a single experimental frame, and it makes computer simulation possible. Hereafter, when we say "model" we mean the lumped model.

The **modeling** relation in Figure 6 represents the validity of the model for reflecting the observed behavior of the real system. The model is *replicatively valid* if it matches some be-

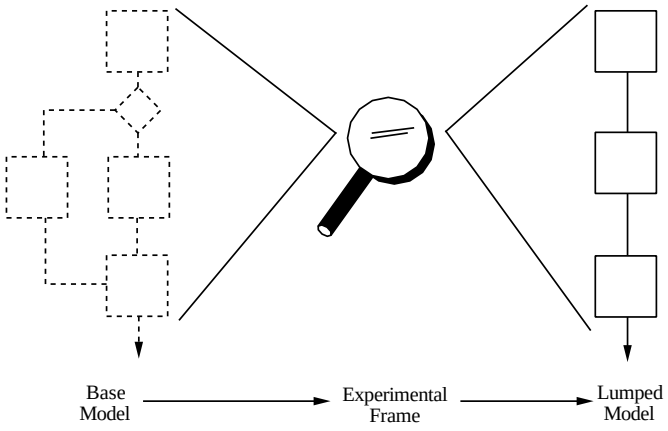

Figure 7: The Modeling Process

havioral data previously acquired from the real system, *predictively valid* if it matches the real system's behavioral data before that data is acquired, and *structurally valid* if it not only reproduces the real system's behavior, but it reflects the way that the real system operated to produce that behavior.

The **computer** may be either a machine or a human. The computer generates the input-output pairs described by the lumped model. The **simulation** relation represents the faithfulness with which the computer carries out the instructions described in the model. To *validate* a model one compares the simulation results with the behavior of the real system. Model validation assumes that the computer realizes the model correctly; otherwise, one may confuse faults in the computer program with faults in the model.

Simulation is an effective technique for validating the model when the real system already exists. If the real system has not been built, then simulation is primarily effective for communicating the proposed system's behavior (i.e., the model's behavior) to the user, and the model must be validated another way. We will discuss one way in Section 4.

3.2 Effective Communication

When we evaluate a model, we actually evaluate two things: the model itself and the description of it. The model itself exists only within the modeler's mind. Others "know" the model through its description; therefore, it is very important that the description be faithful and complete.

According to Zeigler, complete and effective communication of the model includes an informal description, a formal description, presentation of the simulation program, presentation of the simulation experiments and their results, and conclusions about the model's validity. Since the problem domain we're considering in Section 4 lessens the impact of simulation, we'll ignore presenting the simulation experiments and their results for now. The remaining components are listed below.

- **Informal Description**
  - Concept of Operations
  - Assumptions
  - Instructions
- **Formal Description**
  - Computational Framework
  - Instructions
- **Validity Argument**

Note that these items represent only the minimum for effective communication. Other topics, such as a thorough introduction to the model's domain (e.g., the discussion of requirements for military message systems in the SMMS model [16]), are also appropriate and will be valuable for the reader.

3.2.1 The Informal Description

The modeler's progress in defining the model depends on maintaining a clear, overall image of the work. An informal description that presents the essentials — but not the details — helps the modeler maintain this image. However, the informal description's greatest contribution is to facilitate the model's public presentation. The informal description is a natural communication interface that helps the reader comprehend the model. To fulfill these purposes, the informal description should include, in addition to the behavior-generating instructions, the system's operational concept and the assumptions under which the model was constructed.

Concept of Operations

A brief narrative of how the real system operates is useful for understanding the instructions that are presented later. This discussion can present a user's view of the system, i.e., at its external interface, or it can provide a more comprehensive look at the internal workings of the system.

Assumptions

The model describes the behavior of a system that will reside in the real world. Unless the system is completely closed, it will interact with its environment. The assumptions state the modeler's beliefs about the behavior of the system's environment. The assumptions can be viewed as behavior-generating instructions for the environment.

The assumptions are necessary to validate the model. Without them, the model could be interpreted incorrectly. The assumptions should appear early in the informal description, preferably before the instructions, so that the reader is aware of them when reviewing the instructions.

Instructions

The primary purpose of the behavior-generating instructions is to make simulation possible. However, in the informal description, the instructions also provide a forum for working out the details of the computational framework that will be defined for the formal description. The instructions in the informal description are expressed in natural language. A reader who is interested in applying this model to his own problem should be able to infer the model's intended behavior easily — without having to understand a formal notation that would be necessary to simulate it. The instructions are of the form IF x THEN y, where x denotes a condition that must be true for the instruction to apply. The expression y describes the action that will occur if the condition is true.

3.2.2 The Formal Description

The formal description is necessary for simulation, but it also permits the reader to analyze the model rigorously. While the formal description sacrifices the natural communication interface of the informal description, its formal language can facilitate analyzing the model with automated tools. Finally, the formalization helps the modeler detect omissions, inconsistencies and ambiguities in the informal description. Gödel's definition [8] suggests that a specification is formal if it is mechanically checkable. Below we discuss the parts of the formal description of the model.

The Computational Framework

In order to formalize the instructions, the modeler must define a computational framework. The computational framework is more than the language in which the instructions of the model are expressed. It is the description of the automaton on which these instructions are executed. That automaton has a static structure and a dynamic structure. The computational framework should contain only the elements of each structure that are required to express the instructions. Some popular computational frameworks will be discussed more thoroughly in Section 3.3.

Formal Instructions

The instructions of the informal description are restated in the formal notation of the computational framework.

3.2.3 The Validity Argument

The validity of a model (i.e., whether it is replicatively valid, predictively valid or structurally valid) is determined relative to the experimental frame, because the experimental frame determines the scope of the model. It is important to verify that the model is defined under all of the conditions permitted by the experimental frame and that the model is defined only under these conditions. This can be accomplished through a mapping or other correspondence argument between the model and the experimental frame.

There are often factors that are peculiar to a particular modeling effort that the modeler should make explicit. For example, there may be assumptions about the model's domain that are not expressed in the informal description of the model, or the modeler may offer helpful hints about interpreting the model for particular architectures.

3.3 Understanding Computational Frameworks

The computational framework is the specification of the automaton upon which the instructions are executed. Choosing an appropriate computational framework is critical to the model's success. A poor framework yields a model that is difficult to understand and interpret. The choice is especially crucial for security modeling since the model's behavior is validated through analysis rather than simulation.

Abadi and Lamport [1] divide specification techniques into two general classes.

> The popular approaches to specification are based either on states or actions. In a state-based approach, an execution of a system is viewed as a sequence of states, where a state is an assignment of values to some set of components. An action-based approach views an execution as a sequence of actions. These different approaches are, in some sense, equivalent. An action can be modeled as a state change, and a state can be modeled as an equivalence class of sequences of actions. However, the two approaches have traditionally taken very different formal directions. State-based approaches are often rooted in logic, a specification being a formula in some logical system. Action-based approaches have tended to use algebra, a specification being an object that is manipulated algebraically. [1, p. 77]

They believe that the state-based approach is more useful because it discourages oversimplification of the model.

In this section, we will briefly describe both specification techniques. First, we consider the popular state machine approach. Then we examine two types of action-based specifications that have recently shown promise: function call traces and communication event traces. Function call traces describe a system's behavior by stating rules about legal sequences of the system's function calls. Communication event traces state rules about valid sequences of communications events (i.e., inputs and outputs) of the system. The purpose of this section is to give the reader a brief summary of the style of each technique. The reader should examine the cited works for more information. The section will conclude with some important questions for for the developer of the computational framework.

3.3.1 State Machines

The state machine identifies variables that define the state of the system, and it defines a transition function for changing those variables (i.e., for changing the state). The state machine is described fully in many sources (e.g., [2, 15, 19]) but for simplicity, we present only the definition from Arbib [2].

> **Definition** A state machine or *automaton* $M = (X, Y, Q, \delta, \lambda)$, where X is a finite set of *inputs*, Y is a finite set of *outputs*, Q is a set of *states*, δ is the *next-state-function*, $\delta : Q \times X \to Q$, and λ is the *next-output-function*, $\lambda : Q \times X \to Y$.

M is a finite state machine if Q is finite. Because M associates outputs with transitions, it is called a *Mealy machine* [26]. An arbitrary Mealy machine may be replaced by an equivalent machine M' that associates outputs with states, called a *Moore machine* [29], and denoted $M' = (X, Y, Q, \delta, \beta)$, where $\beta : Q \to Y$. These machines are equivalent if we let $\lambda = \beta \circ \delta$. In our discussion, we shall treat all machines as Moore machines.

For any machine M, Q represents all possible states of the machine. Given the set of inputs, X, and an initial state, q_0, δ determines which states in Q are reachable. Any state, q_i, is distinguishable from any other state, q_j, by the values of its outputs, $\beta(q_i)$ and $\beta(q_j)$, respectively. We may interpret $\beta(q_i)$ as the values of the system's variables in state q_i, and we may interpret X as the set of commands that can modify those values. The next-state function δ represents the behavior-generating instructions.

Communication of the state machine should include

- the inputs X, e.g., the commands of the system,
- the outputs Y,
- how a state $q_i \in Q$ is represented, i.e., the variables that describe the state,
- the initial state q_0 of the system, i.e., the values of the descriptive variables initially,
- the output function β, which may be several functions, and
- the state transition function δ.

Two machines M and M' are equivalent *iff*

$$\{M_q \mid q \in Q\} = \{M'_{q'} \mid q' \in Q'\},$$

i.e., their input/output functions are equivalent.

3.3.2 Function Call Traces

Function call trace specifications [3, 20, 21], or Parnas traces, are similar to algebraic specifications [13]. According to [24], a Parnas trace specification consists of a syntax section that lists function names and types for a software module and a semantics section that specifies the observable behavior of that module. The behavior is described by assertions on sequences of function calls. The assertions are constructed from logical connectives, the function calls and their parameters, and two predicates L and V. L is true when applied to a legal trace, and V specifies the value returned by a legal trace ending in a function call.

Consider a simple stack module [21] that permits multiple pushes and pops. The syntax portion of the specification consists of three operations.

SYNTAX

PUSH: $O \ldots O$
POP: $[I]$
TOP: $\rightarrow O$

The token I denotes the integers, and O represents the objects to be pushed and popped. *PUSH* accepts any number of objects. *POP* can be parameterless or it can accept an integer representing the number of objects to pop. *TOP* returns a single object.

The semantics section describes the behavior of the module.

SEMANTICS

(1) $L(T) \rightarrow L(T.\mathit{PUSH}(o))$
(2) $L(T.\mathit{TOP}) \leftrightarrow L(T.\mathit{POP})$
(3) $T.\mathit{PUSH}(o_1, \ldots, o_n) \equiv T.\mathit{PUSH}(o_1).\mathit{PUSH}(o_2, \ldots, o_n)$
(4) $T.\mathit{POP} \equiv T.\mathit{POP}(1)$
(5) $i > 1 \rightarrow T.\mathit{POP}(i) \equiv T.\mathit{POP}.\mathit{POP}(i-1)$
(6) $T \equiv T.\mathit{PUSH}(o).\mathit{POP}$
(7) $L(T.\mathit{POP}) \rightarrow T \equiv T.\mathit{TOP}$
(8) $L(T) \rightarrow V(T.\mathit{PUSH}(o).\mathit{TOP}) = o$

The variables i and o range over integers and objects, respectively, the "." is read "followed by", and all other symbols have their conventional meaning. In short, the semantics section states that the behavior of any legal trace consisting of *PUSH*s, *POP*s and *TOP*s can be inferred from these rules.

Two traces are equivalent, $T \equiv S$, *iff* for any trace R, $L(T.R)$ *iff* $L(S.R)$ and for non-null R, $V(T.R) = V(S.R)$, if defined. Intuitively, equivalence means that the traces $T.R$ and $S.R$ are indistinguishable in their future behavior with regard to L and V.

3.3.3 Communication Event Traces

Communication event trace specifications [14, 33], or Hoare traces, describe the communications behavior between sequential processes executing concurrently. As a result, this computational framework may be applied more effectively at a higher level of abstraction than either state machines or Parnas traces because it is well suited for concurrency (unlike state machines) and because it does not rely on a knowledge of the software module structure (unlike Parnas traces). However, this expressive power limits its utility at lower levels of abstraction, e.g., the specification of a single, sequential software process, so the evaluator may expect Hoare traces to be used in conjunction with another computational framework.

Hoare traces are sequences of communication events, i.e., inputs and outputs, of a process. A process P has an alphabet αP that determines the communication events of P. Each event is of the form $ch.v$, where ch names a channel over which P communicates with other processes and v is a value that can pass over that channel. P is described by the set of traces T of events in which P can engage.

To better understand their use, consider the simple example of the RS-232 repeater [28]. Briefly, the repeater has an input data line, an output data line and an error line. Valid characters received over the input data line are transmitted after some delay over the output data line. Invalid characters are transmitted over the error line. Every valid character must be transmitted over the output data line, only received characters are transmitted, and valid characters are transmitted in the order they are received. These requirements make this application a good candidate for Hoare traces since the traces preserve the order of inputs and outputs.

We introduce IN and OUT as the input data line and output data line, respectively, $\checkmark$ as the process termination event, $\downarrow$ as the trace selection operator and $\langle\rangle$ as the empty trace. Given a trace t and the channel OUT, $t \downarrow OUT$ returns the sequence of values transmitted over OUT. The function *Valid* takes a sequence of characters and returns only those characters that are valid.[3] The function *Last* takes a trace and returns the last element. The repeater must satisfy the requirement *ValidRelay*.

ValidRelay$(t1)$ if $t1 = \langle\rangle$ then true
elseif Last$(t1) = \checkmark$
then $(t1 \downarrow OUT)$ = Valid $(t1 \downarrow IN)$

The variable $t1$ represents a trace in which the repeater can engage, and *ValidRelay* must hold for all such traces.

Hoare traces are a powerful specification tool for certain types of requirements. The reader should examine the cited works for more information.

3.3.4 Determining the elements

Once the computational framework is selected, the modeler must determine, by examining the requirements, what elements should be specified in the framework, i.e., what the framework will look like. Zeigler identifies six questions for the modeler.

[3] We will ignore what makes the characters valid for now.

1. What will be the model's time base (system clock)? Does the clock advance periodically, such that changes (events) occur only at the clock transitions, or does the clock advance smoothly and continuously? If the latter, do events occur in discrete jumps or continuously?

   The modeler must first decide *if* the time base needs to be modeled. While a computer system can be modeled as a discrete time system (after all, the computer's clock advances periodically), computer systems are typically viewed as continuous time/discrete event systems, and the time base is not modeled explicitly.

   If the time base is significant, however, then the framework must include the system's clock (or a similar entity) as a descriptive variable (for state machines), a function call (for Parnas traces) or a communication event (for Hoare traces). For example, continuous time/continuous event systems are often modeled as differential equations.

2. Are the model's values represented discretely, continuously or some combination of the two?

3. Will random values be used in the model? Models with no random values, e.g., those whose behavior is deterministic, are relatively straightforward to analyze and are the most common. However, recent research [30, 10, 11, 12] has demonstrated the merits of probabilistic models for security-critical systems.

4. Will the model interact with its environment? For security models, the answer is almost always affirmative. This quality underscores the importance of identifying as many assumptions as possible in the informal description.

5. Will the model be influenced by history? Almost every security model of interest is influenced by history. For state machines, history must be represented within the definition of state. This can be a difficult task for the modeler, because he must determine what is historically relevant. The task is simpler for trace-based models, where history is represented by the entire trace.

4 Modeling Security

We learned in the last section that a model is a set of instructions for generating the behavioral data of a real system. The term *security policy model* implies that we want to model the COMPUSEC policy. However, the COMPUSEC policy does not (generally) exhibit behavior. It describes how the behavior of the real system must be *constrained*. Consequently, an SPM is a set of instructions for generating the behavioral data of a real system[4] that is constrained by the COMPUSEC policy.

Recall from Section 2.2 that the COMPUSEC policy identifies the assertions to be enforced by the COMPUSEC countermeasures, as well as the assumptions on which those assertions are based. In Section 2.3, we discussed how the COMPUSEC policy and the computer functional requirements are combined to derive the assertions (and perhaps identify more assumptions) for the SPM. These assertions must be enforced by the architecture that will also satisfy the computer functional requirements. Since the SPM generates the behavior of the proposed system through its instructions, and since this behavior must satisfy the constraints of the assertions, the instructions of the SPM must be demonstrated to satisfy the assertions of the SPM. We do so by including the assertions in the SPM and demonstrating that the instructions obey them. We also include the assumptions for clarity.

Figure 8 interprets Figure 7 ("The Modeling Process") for modeling security. As before, we hypothesize the base model by examining the functional requirements that the system must satisfy. The COMPUSEC policy becomes the experimental frame from which we view the base model. The resulting lumped model, the SPM, defines the critical requirements for the system, and it is valid only for those functional requirements and that COMPUSEC policy.

Effective communication of the SPM (listed below) must include the assertions and the assumptions. The informal description expresses the assertions in the context of the user's view of operations. The formal description restates the assertions within the computational framework. Since the instructions and the assertions are both stated formally, we can prove that the instructions satisfy the assertions.

- **Informal Description**
  - User's View of Operation

[4] For TCSEC evaluations, the *real system* is the trusted computing base (TCB).

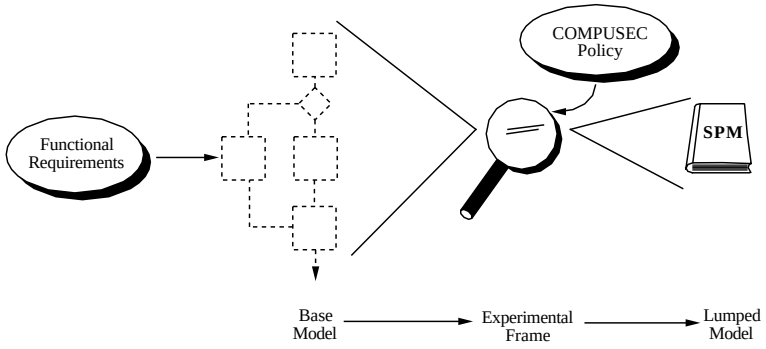

Figure 8: The Security Modeling Process

  - Assumptions
  - Assertions
  - Instructions

- **Formal Description**
  - Computational Framework
  - Assertions
  - Instructions

- **Validity Argument**

4.1 How the SPM is Different from Ordinary Models

This section discusses how security affects the communication of a model. In particular, we review the concept of operations, the assumptions, the computational framework, the instructions and the argument for validity.

4.1.1 User's View of Operations

In Section 3.2.1, we identified the *concept of operations* as part of the informal description of the model. For security models, we adopt the approach of the SMMS model [16] which contains a *User's View of Operations* that describes the system at its external interface. This view is more appropriate for an SPM since security is usually defined at the external interface of the system.

The user's view of operations is a forum for discussing the system's complete behavior, including non-security-relevant behavior. Such discussion will help the reader appreciate the context in which the security constraints are being applied. It will also help the evaluator assess the user's interaction with the system to determine whether appropriate constraints are defined.

4.1.2 Assumptions

In addition to assumptions from the COMPUSEC policy that are necessary to validate the SPM, the developer may include assumptions about the system's implementation that simplify the SPM. For example, if an off-the-shelf cryptographic chip will be used, then assumptions about the existence and correct operation of that chip are appropriate. For trusted systems, an analogous comment can be made about the role of the trusted products upon which the system is based (more about this in Section 5).

4.1.3 The Computational Framework

Writing the critical requirements in the informal description of the SPM is straightforward. However, restating these requirements formally within a computational framework is less trivial. Below we revisit the computational frameworks discussed in Section 3.3 and discuss how the constraints should be expressed within them.

Using state machines. For state machines, we want to constrain the state, i.e., the values that its variables can have, and the transitions between states. Constraining just the states or just the transitions is not sufficient [22]. Security is demonstrated inductively for state machines: if the machine begins in a secure state, and if all reachable states are secure, then the machine is secure.

To make our task easier, we recognize that for secure systems, we are not concerned with the values of all system variables but just the security-relevant ones. Consequently, we define the *state* explicitly as the cross-product of all security-relevant variables. To illustrate this point, we present the simple state machine model described by McLean [23].

The system is a quadruple, $\Sigma = (V, R, T, v_{init})$, where V is the set of states, R is a set of requests or commands to change the state, T is the transition function, and v_{init} is the system's initial state, and $v_{init} \in V$.

We identify the security-relevant variables of the system:

- S is the finite set of subjects, or active entities, of the system. Subjects represent users and the processes acting on their behalf.
- O is the finite set of objects, or passive entities, of the system. Objects represent files and other storage entities.
- L is the finite set of security levels that can be assigned to subjects and objects. The set is ordered by the relation $\prec$, or predecessor. In our simple example, the set is totally ordered, but in practice, the set could resemble a lattice (c.f. [4]).
- $A = \{\text{read, write, execute}\}$ is the finite set of accesses that a subject may have to an object.
- $B = \mathcal{P}(S \times O \times A)$ is the set of all possible access sets, where $\mathcal{P}$ is the powerset function.
- $F = (f_s, f_o)$, where $f_s : S \to L$ and $f_o : O \to L$, represents all possible security levels of subjects and objects, respectively.

Note that by examining $b \in B$, f_s and f_o, we can determine the values of the other variables, so it is sufficient to state that $V \subseteq B \times F$. The transition function, $T : S \times R \times V \to V$, moves Σ to a new state given a subject's request in the old state. Finally, we declare an auxiliary constant function, $C = (c_s, c_o)$, where $c_s : S \to \mathcal{P}(S)$ and $c_o : O \to \mathcal{P}(S)$, to represent all the subjects that can modify a particular subject's clearance and all the subjects that can modify an object's classification, respectively.

Constraints on states are assertions on the values of b, f_s and f_o. For example, McLean asserts that a state $v = (b, f)$ is *simple secure* if for each element b of the form (x, y,read), $f_s(x) \geq f_o(y)$. In other words, if a subject x has "read" access to an object y in some state v, then the clearance of x must be greater than the classification of y.

Constraints on state transitions are assertions on the values of the state variables in a new state, $v' \in V$, given the values of those variables in an old state, $v \in V$, where there exists some $s \in S$ and $r \in R$ such that $T(s, r, v) = v'$. For example, the state transition assertion might state that some relationship must exist between $f_o(o)$, where $o \in O$, in the new state and $f_o(o)$ in the old state (e.g., the classification of object o must not change).

Using Parnas Traces. To constrain Parnas traces, the developer states explicitly, using the legal and value functions, L and V, what a legal trace is and what values it may return according to the constraints. If necessary, the developer must add new rules to the semantics section to express the constraints. See McLean [24] for examples.

McLean [24] has specified a definition of security using Parnas traces, but the framework has not been applied to an operational system. For an interesting comparison of a system modeled using both a state machine and Parnas traces, read Landwehr, Heitmeyer and McLean [16] and Cross [5].

Using Hoare Traces. Like Parnas traces, the developer using Hoare traces must define the set of acceptable traces; however, in this case, the exercise involves constraining the presence and order in the trace of security-sensitive communication events. For an example, read Payne, Moore, Mihelcic and Hayman [36].

4.1.4 The Instructions

A significant debate exists about whether the behavior-generating instructions should be included in the SPM. McLean [24] notes,

> [T]here are two related, but distinct, senses of the term *security model* [. . .] In the more limited use of the term, a security model specifies a particular mechanism for enforcing confidentiality [. . .] In the more general usage of the term, security models are specifications of a system's confidentiality requirements and are not 'models' at all in that they specify security requirements without describing any particular mechanism for implementing these requirements.

The "particular mechanism" to which McLean refers are the behavior-generating instructions. Since the state machine framework is prominent within the security community, these instructions are often called *transition rules*. Millen [27] refers to SPMs with transition rules as *concrete*, because they are specific to a particular architecture. Similarly, SPMs without the rules are called *abstract*.

Historically, the concrete SPM is preferred. The popular SPM by Bell and LaPadula [4][5], for example, has eleven transition rules. In general, transition rules simplify the interpretation of the SPM for a particular architecture. However, McLean's point is that the SPM can satisfy its purposes without transition rules. The modeler can instead state properties that must be true of all transition rules, so he is freed from making design decisions prematurely. However, the modeler may err interpreting these properties for more concrete structures, so there is some tradeoff.

The SMMS SPM [16] is an abstract SPM. It expresses constraints on states and constraints on an abstract state transition function. Unlike the Bell and LaPadula SPM, which embeds the state transition constraints within its transition rules, the SMMS SPM makes its state transition constraints explicit and defers the specification of the transition rules. Security-critical functions of the SMMS are demonstrated formally to satisfy the state transition constraints.

The SPM of Gove, Williams and Jaworski [9] combines the approaches of Bell and LaPadula and the SMMS authors. It includes transition rules (or at least it is designed to include

[5] The Bell and LaPadula SPM inspired the TCSEC's modeling requirements.

them), but it also defines explicit state transition constraints that the rules must satisfy. The state transition constraints, together with the state constraints, make the definition of security explicit, and the transition rules simplify the interpretation of the SPM.

The latter approach may be the most practical for modeling trusted systems. Since the definition of security is usually complex for these systems, it is prudent to specify it without making the design decisions that would be necessary to express the instructions. On the other hand, once the definition is expressed, the instructions form a convenient bridge for completing the interpretation of the model to the top-level design specification.

4.1.5 The Validity Argument

Unlike the models addressed by Zeigler, which are for natural systems that can be measured, the SPM models the behavior of an artificial system (the computer) that *does not exist yet*. Nevertheless, we can interpret Zeigler's definition of *validity*, i.e., replicatively valid, predictively valid or structurally valid, for the SPM. If the SPM generates outputs (in simulation) that, given the inputs, are consistent with the trusted system's critical requirements, and if the trusted system is built to also generate those outputs, then the SPM is *predictively valid*. If the trusted system is built to perform exactly as the model specifies (i.e., the trusted system's transformations are identical to the SPM's), then the SPM is *structurally valid*. The assurance argument determines predictive or structurally validity. Because the trusted system does not exist, the SPM cannot be replicatively valid.

The SPM must be valid for its experimental frame, i.e., the COMPUSEC policy. This means that the SPM must model *all* of the relevant requirements of the COMPUSEC policy and *only* those requirements. (The relevance of a particular requirement was discussed in Section 2.5.) The most straightforward approach is to demonstrate through a two-way mapping that the critical requirements are derived completely and exclusively from the relevant requirements of the COMPUSEC policy.

To complete the overall assurance argument for the system, it is also necessary to demonstrate that the assertions of the formal description correspond to the assertions of the informal description. This demonstration should be performed as the formal assertions are specified.

Finally, if instructions are included in the SPM, the TCSEC states that the formal instructions must be shown formally through proof to satisfy the assertions.

5 The Trusted Product SPM

Trusted systems usually contain trusted products. Trusted products are evaluated against the TCSEC and awarded a rating such as B1 or A1. This rating reflects not only the security mechanisms that the trusted product uses but also the assurance that those mechanisms function as intended. The trusted system relies on the product's features and assurance for its correct, secure operation.

Typically, the product enforces only a portion of the COMPUSEC policy. The developer is left with three choices for enforcing the remaining requirements.

1. Develop the trusted system from scratch, i.e., ignore the trusted product. This choice is almost never feasible for economic reasons.

2. Develop the trusted system on a product that enforces some requirements and modify the product as necessary to enforce the remaining requirements. This choice, like the first, usually requires a complete re-evaluation of the product and can be very expensive. In addition, the developer is challenged to integrate the documentation for the unmodified portions of the product into the new system documentation.

3. Develop the trusted system on a product that enforces some requirements and provide enforcement for the remaining, application-specific requirements in a trusted component that is completely external to the trusted product, i.e., do not modify the product.

Choice 3 is the most attractive if it is feasible. It is very similar to the concept of TCB subsetting described by Shockley and Schell [38]. First, this choice usually yields the shortest development and evaluation times for the system, because a significant part of the system, i.e., the product, is already developed and evaluated. Second, the product provides a well-defined interface for the design of the application-specific component, and documents prepared for the product (such as planning documents and design documents) can provide guidance for writing similar documents for the application-specific component. Third, the system developer is relieved from demonstrating in the model that certain COMPUSEC requirements, i.e., those enforced by the product, are properly enforced by the system.

However, the developer must confirm that the security policy enforced by the product is compatible with the COMPUSEC policy to be enforced by the system.[6] Otherwise, it may be necessary to "turn off" the mechanisms in the product that enforce additional policies. This kind of modification may require re-evaluation of the product. Ensuring compatibility between the system and product security policies should occur while the product is still under evaluation.

Guidelines for writing the trusted system's SPM, given the trusted product's SPM, are provided below. These guidelines assume that development choice 3 is being used. In other words, the developer is building the trusted system using a trusted product, and the product will not be modified. All application-specific requirements will be enforced in a trusted component that sits "on top of" the trusted product.

[6] In most cases, "compatibility" is achieved if the system's COMPUSEC policy is a *restriction* of the products security policy.

1. Compare the trusted system's COMPUSEC policy with the trusted product's security policy. If the COMPUSEC policy is compatible with the product's policy, then an application-specific trusted component can be built on this product. Otherwise, either hide the differences in the component or choose another product.

2. Write the SPM, both the informal and formal descriptions, for the trusted system. Try to keep the computational framework for the trusted system's SPM consistent with the computational framework for the trusted product's SPM.

3. Map the computational framework of the system's SPM to the computational framework for the product's SPM. This mapping demonstrates, in an abstract way, how the system will "use" the product. For example, how will users, or their representation to the system, be implemented by the product? Such information is crucial for a successful design.

4. Compare the system SPM's formal security assertions to the product SPM's formal security assertions using the mapping. This comparison should identify exactly what the application-specific trusted component must enforce and what the trusted product will enforce.

5. Reclassify the assertions that the product will enforce as assumptions for the trusted system.

This approach is illustrated in [37].

6 The Evaluation

The evaluator should assess the SPM's communication (see Section 4) against the four quality requirements identified in Section 2.5 (completeness, correctness, consistency and clarity). Then the SPM should be examined to determine if it satisfies the purposes described in Section 2.4. This latter review should include an assessment of the SPM's definition of security.

A comprehensive evaluation of the SPM's communication requires knowledge of the system's mission. A simple checklist cannot be constructed for all cases. However, we can provide some motivating questions for the evaluator. These questions are only a starting point.

- **The Informal Description**
  - **Concept of Operations:** Is the behavior described here consistent with the intended behavior of the system as identified in the COMPUSEC functional requirements? Have all of the security-relevant, user-visible operations been identified? Is the description clear enough for the user to understand the role he or she plays in the overall mission and how the trusted system constrains that role?
  - **Assumptions:** Are all assumptions from the COMPUSEC policy identified? If no assumptions are stated in the COMPUSEC policy, then the developer should attempt to state the assumptions about the other INFOSEC disciplines here. Are all assumptions about the system's implementation identified? For example, if a trusted product will be used, are all of the assumptions about the security behavior of that product noted?
  - **Assertions:** Can the assertions stated here be derived from the COMPUSEC policy? Is the definition of security expressed by these assertions reasonable for this trusted system and this mission?
  - **Instructions:** If included (see Section 4.1.4), is the behavior that they describe consistent (or at least more restrictive) than the behavior allowed by the COMPUSEC functional requirements? The evaluator may also assess here whether the behavior described by the instructions satisfies the assertions.
- **The Formal Description**
  - **Computational Framework:** Is the computational framework selected appropriate for this system and mission? Have all relevant elements of the framework been defined? Is the framework itself consistent and clear?
  - **Assertions:** Are these assertions a complete and correct formal restatement of the assertions of the informal description? Are they stated correctly within computational framework?
  - **Instructions:** If they are included, are these instructions a complete and correct formal restatement of the instructions stated in the informal description? Are they stated correctly within the computational framework?
- **The Validity Argument**

Is the derivation of the COMPUSEC policy to the SPM assertions and assumptions discussed here? Is the derivation complete and correct? Are the formal instructions proven to satisfy the formal assertions?

References

[1] Martin Abadi and Leslie Lamport. Composing specifications. *Transactions on Programming Languages and Systems*, pages 73–132, January 1993.

[2] Michael A. Arbib. *Theories of Abstract Automata*. Prentice-Hall, 1969.

[3] W. Bartussek and D. L. Parnas. Using traces to write abstract specifications for software modules. Technical Report TR 77-012, University of North Carolina, Chapel Hill, N. C., December 1977.

[4] D.E. Bell and L.J. La Padula. Secure computer system: Unified exposition and Multics interpretation. Mitre Technical Report MTR-2997, Mitre Corp., Bedford, MA, March 1976.

[5] C. B. Cross. A trace specification of the MMS security model. NRL Memorandum Report 6216, Naval Research Laboratory, Washington, D. C., July 1988.

[6] Department of Defense. *DoDD-STD-2167A, Defense System Software Development*, February 1988.

[7] J.N. Froscher, J.P. McDermott, C.N. Payne, and H.O. Lubbes. Successful acquisition of certifiable application systems (or: How not to shake hands with the tar baby). In *Proc. Sixth Annual Computer Security Applications Conference*, Tucson, AZ, December 1990. IEEE Computer Society Press.

[8] K. Goedel. On formally undecidable propositions of the principia mathematica and related systems. In M. Davis, editor, *The Undecidable*, pages 5–38. Raven Press, Hewlett, N. Y., 1965.

[9] R.A. Gove, Lisa Jaworski, and James G. Williams. To Bell and back: Developing a formal security policy model for a C^2 system. In *Proc. Seventh Annual Computer Security Applications Conference*, San Antonio, TX, December 1991. IEEE.

[10] James W. Gray, III. Probabilistic interference. In *Proc. 1990 IEEE Symposium on Security and Privacy*, Oakland, CA, May 1990.

[11] James W. Gray, III. Probabilistic interference in restrictive systems. NRL Report 9315, Naval Research Laboratory, Washington, D.C., June 1991.

[12] James W. Gray, III and Paul F. Syverson. A logical approach to multilevel security of probabilistic systems. In *Proc. 1992 IEEE Symposium on Research in Security and Privacy*, Oakland, CA, May 1992. IEEE.

[13] J. Guttag. Abstract data types and the development of data structures. *Comm. ACM*, 20:396–404, June 1977.

[14] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, UK, LTD., London, 1985.

[15] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[16] C. Landwehr, C. Heitmeyer, and J. McLean. A security model for military message systems. *ACM Transactions on Computer Systems*, 2(3):198–222, August 1984.

[17] Carl E. Landwehr. The best available technologies for computer security. *IEEE Computer*, pages 86–100, July 1983.

[18] Carl E. Landwehr. Some lessons from formalizing a security model. In *Proc. Verkshop III* in *ACM Software Engineering Notes.* Association for Computing Machinery, August 1985.

[19] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the Theory of Computation.* Prentice-Hall, 1981.

[20] John McLean. A formal foundation for the trace method of software specification. NRL Report 4874, Naval Research Laboratory, September 1982.

[21] John McLean. A formal method for the abstract specification of software. *J. ACM*, 31(3):600–627, July 1984.

[22] John McLean. A comment on the "Basic Security Theorem" of Bell and LaPadula. *Information Processing Letters*, 20(2):67–70, February 1985.

[23] John McLean. The algebra of security. In *Proc. 1988 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, April 1988.

[24] John McLean. Proving noninterference and functional correctness using traces. *Journal of Computer Security*, 1(1), 1992.

[25] John McLean. *Encyclopedia of Software Engineering*, chapter Security Models. John Wiley & Sons, February 1994.

[26] G.H. Mealy. A method for synthesising sequential circuits. *Bell System Tech. J.*, XXXIV:1045–79, 1955.

[27] Jonathan K. Millen. Models of multilevel computer security. Mitre Technical Report MTR-10537, The Mitre Corporation, January 1989. Also in *Advances in Computers*, Vol. 28, Academic Press.

[28] Andrew P. Moore. Using CSP to develop trustworthy hardware. In *Proc. Fifth Annual Conference on Computer Assurance*, Gaithersburg, MD, June 1990. National Institute of Standards and Technology.

[29] E.F. Moore. *Automata Studies*, chapter Gedanken–Experiments on Sequential Machines, pages 129–53. Princeton University Press, Princeton, NJ, 1956.

[30] Ira S. Moskowitz. Quotient states and probabilistic channels. In *Proc. The Computer Security Foundations Workshop III*, pages 74–83, Franconia, NH, June 1990.

[31] National Computer Security Center, Ft. Meade, MD. *DoD 5200.28-STD, Trusted Computer System Evaluation Criteria*, December 1985.

[32] National Computer Security Center. A guide to understanding security modeling in trusted systems. Technical Report NCSC–TG–10, National Security Agency, October 1992.

[33] E.R. Olderog and C.A.R. Hoare. Specification oriented semantics for communicating sequential processes. *Acta Inform.*, 23:9–66, 1986.

[34] David L. Parnas, A. John van Schouwen, and Shu Po Kwan. Evaluation of safety-critical software. *Communications of the ACM*, 33(6), June 1990.

[35] Charles N. Payne, Judith N. Froscher, and Carl E. Landwehr. Toward a comprehensive INFOSEC certification methodology. In *Proceedings of the 16th National Computer Security Conference*, pages 165–172, Baltimore, MD, September 1993. NIST/NSA.

[36] Charles N. Payne, David M. Mihelcic, Andrew P. Moore, and Kenneth J. Hayman. The ECA critical requirements model. NRL Formal Report 9528, Naval Research Laboratory, Washington, D.C., December 1992.

[37] C.N. Payne, J.N. Froscher, and J.P. McDermott. On models for a trusted application system. In *Proc. Sixth Annual Computer Security Applications Conference*, Tucson, AZ, December 1990. IEEE Computer Society Press.

[38] William R. Shockley and Roger R. Schell. TCB subsets for incremental evaluation. In *AIAA/ASIS/IEEE Third Aerospace Computer Security Conference: Applying Technology to Systems*, pages 131–139, Washington, D.C., December 1987. American Institute of Aeronautics and Astronautics.

[39] Daniel F. Sterne. On the buzzword 'security policy'. In *Proc. Symposium on Research in Security and Privacy*. IEEE, June 1991.

[40] Bernard P. Zeigler. *Theory of Modelling and Simulation*. Wiley, New York, NY, 1976.